A Basic HTTP server

rdoss.com

April 26, 2001

1 Abstract

With the advent of the Hyper Text Transport Protocol (HTTP), software application development has been profoundly influenced. Many applications can benefit from the existing Web infrastructure, including the popularity of Web browsers, HTML, XML, Javascript and other Web languages. This project focuses on the HTTP protocol and presents an extensible framework which can support HTTP, as well as other protocols. Developing a framework for which TCP/IP servers can be implemented, as well as implementing a portion of the HTTP protocol. The protocol itself, along with the RFC's that describe it have been studied in great detail, and described formally in Backus Naur Form (BNF). The framework for which TCP/IP servers can be implemented utilizes Object Orientated software development concepts and can be easily customized to support a wide variety of servers as well as research in communication protocol development.

2 Keywords and Phrases

Byte	Unit of computer memory, 1 byte equals 8 bits
CGI	Common Gateway Interface
CPU	Central Processing Unit
Cookies	Data stored on HTTP clients, used to preserve state
Cracker	Criminal Hacker, one who misuses computers
Gig	One Gigabyte
Gigabyte	Unit of computer memory, 1 Gigabyte equals 1 billion bytes
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transport Protocol
Hacker	Advanced computer user
IP	Internet Protocol
Javascript	Scripting Language used in conjunction with HTML
K	One Kilobyte
Kilobytes	Unit of computer memory, 1 kilobyte equals 1000 bytes
Megabyte	Unit of computer memory, 1 Megabyte equals 1 million bytes
Meg	One Megabyte
Mhz	Megahertz
MIME	Multipurpose Internet Mail Extensions
Process	Unit of work performed by the system
Posix	Portable Operating Systems Interface for UNIX
RISC	Reduced Instruction Set Computer
RPM	Revolution Per Minute
SCSI	Small Computer Systems Interface
SDRAM	Static-Dynamic Random Access Memory
SMTP	Simple Mail Transport Protocol
SSL	Secure Socket Layer
Server	Computer unit shared among users
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
WWW	World Wide Web
dynamc	Refers to data whose state may change
thread	Point of execution or control within a process
RFC	Request for Comments
static	Refers to data whose state shall not change

Contents

3 Introduction

With the advent of the Hyper Text Transport Protocol (HTTP) and the Hyper Text Markup Language (HTML), software engineering has been profoundly influenced. The HTTP protocol forms the foundation of the World Wide Web (WWW) and HTML is the language in which documents available via the WWW, called web pages, are authored. In most courses of study, the actual protocol implementation is not considered. However, the HTTP protocol has been shown to be functionally capable of supporting a myriad of different applications and is a rich area of study.

The protocol is usually implemented at the application layer of the TCP-IP network model. Understanding how it functions brings forth many benefits to students. In particular, the protocol presents many challenges to implementation, some of these include performance, multi-processing, multi-tasking, memory management, interfacing with the underlying host operating system, TCP-IP application programming, and understanding of the Request for Comments that describes the protocol. How TCP-IP can be used to support transactions is also of importance.

While much of the work is beyond the scope of a semester project, it is possible for one to implement a minimal system which allows the student to investigate almost all aspects of writing a UNIX daemon. Further, it allows for others to explore the server, improving upon it, and developing APIs to utilize the protocol for building applications. Having a basic implementation may allow another student to develop custom APIs for application providers to use. Other students may examine performance, or explore portability issues, while others can use the basic ideas and concepts to develop new or related protocols or research new features and/or enhancements to the existing protocol.

The HTTP protocol is a rich and commercially viable system, especially with upcoming applications to embedded systems, streaming media and digital libraries. In the near future, smart devices with Internet capabilities will be available to consumers providing various different services using HTTP among other protocols.

3.1 Problem Statement

To selectively implement portions of the HTTP protocol so as to provide a basic functioning system. The intent is to learn how this protocol functions

internally, and gain much needed knowledge in a system that affects many aspects of technology and society, including e-commerce. Additional knowledge gained will be the understanding and working with the Internet Request for Comments system of documentation. Incident to the implementation of the protocol, will be the creation of a framework for developing TCP/IP based servers.

3.2 Background and Previous Work

Older versions of the protocol, versions 0.9 and 1.0 of the protocol are described by Berners-Lee et. al. [7].

The most current version 1.1 of the HTTP protocol is described by Fielding et. al. [10]. Access Authentication for HTTP is described by Franks et. al. [11]. HTTP state management is described by Kristol and Montulli [9].

The standard version 2.0 of HTML is described by Berners-Lee and Connolly [1]. The specification for HTML forms is described by Nebel and Masinter [15]. The specification for HTML tables is described by Raggett [17].

URLs are described by Berners-Lee et. al. [12]. Storage of URLs in the Domain Name System is described by Daniel and Mealling [13].

Multipurpose Internet Mail Extensions (MIME) are described by Freed and Borenstein [2, 3, 4], K. Moore [14] and Freed et. al. [16].

The secure socket layer is described by Freier et. al. [8].

3.3 Overview about the Project

3.3.1 Overview of HTTP

Reliability The protocol functions on the Application Layer of a given network architecture. It requires a reliable, connection based transport mechanism and does not provide for reliability on its own. The protocol itself is often implemented as part of the application layer of TCP/IP, although it need not be[5].

Requests The protocol functions under the client-server model, wherein clients make requests to the server, and the server responds to the client[1]. All transactions in HTTP are described as being requests and responses. The requests are referred to as "methods", resources that are requested are referred to as "objects". This terminology was used to keep in line with object-orientated methodologies[18]. [2] The requests themselves are specified in all capital letters, and the protocol is case-sensative with this respect. Some possible requests are the following:

Method	Description
GET	Request for a web page
HEAD	Request for a web page's header
PUT	Request to store a web page
POST	Append to a resource
DELETE	Remove a web page
LINK	Connect two existing resources
UNLINK	Disconnect two resources

Many of the methods where defined as HTTP developed. For example, originally, a version 0.9 compatible server only provided a GET method to obtain resources. Version 1.0 provided GET, HEAD, and POST methods, with other methods being options. Version 1.1 defines all of the above methods, and additional ones for multiple gets, and so on.

State The protocol is stateless. This means that for every request made, the data must be "self-contained". Hence, a server is not required to maintain any state information regarding a transaction[5].

Traffic Flow The protocol is bi-directional in its data flow. A client may request a resource from the server and it may also provide the server with information. An example of this is a POST request, where data is given to the server by the client as is normally done when using the Common Gateway Interface (CGI).

Compatibility The protocol is quite flexible, and thus has many possible settings. A given client and server may negotiate among those settings using accept headers so as to ensure compatibility.

[1]For the purpose of this document, the terms client, agent and web browser will be used interchangeably

[2]For the purpose of this document, the terms requests and methods will be used interchangeably, as well as resources and objects.

Caching The protocol allows for web browsers to cache previously requested resources so as to reduce overhead. A given browser uses conditional requests to assert that the most recent copy of a requested resource is stored in its cache, otherwise, the cache is updated.

Further, the architecture allows for intermediary entities to cache resources and respond to client requests. Such an entity, called a "proxy server" [3]. reduces the processing load imposed on the server. When a client makes a request, the proxy server cache from resources on the server. A server where original copies of resources are stored is called an "origin server".

3.3.2 URLs

Uniform Resource Locator (URL) [4] is a naming scheme that allows for the unique identification of a web page or resource available via the web. It is a specific type of Uniform Resource Identifier (URI) that is composed of the following components:

1. Protocol
2. Hostname or Internet Protocol (IP) address
3. Optional port, default is port 0x80
4. Path, whose components are separate via a slash
5. Optional parameters specified by the client
6. Optional query string

The general HTTP URL form can be expressed as:

```
http://hostname|ip address[:port]/path[;parameters][?query]
```

All web pages are assigned a URL. A client specifies a URL when performing a request. The client and server are responsible for composing/decomposing the URL. The protocol indicates what protocol the client and server are to use[5].

[3] Proxy servers are referred to in this document for completeness. Their implementation is outside of the scope of the project

[4] URLs vary slightly with respect to protocol version. Versions 0.9 and 1.0 do not have a parameters component.

[5] For the purpose of this document, we deal with only HTTP; however, a URL maybe used to indicate another protocol, such as FTP.

The hostname is resolved to an IP address using the Domain Name System (DNS). The port specifies which port the client should connect to. The path is mapped by the server to a valid path on its host file system, usually done by pre-appending the server's root directory to the path specified in the URL. For example, if a user asks for *http://hostname/index.html* and we have a server running on *hostname* whose root directory is */usr/local/httpd/htdocs/* the server may resolve the path *index.html* to */usr/local/httpd/htdocs/index.html*. The parameters are provided by the client. The query is used to provide a name value pair of the form:

```
?name=value{\&name=value}
```

This component of the URL in a GET method that uses CGI, is stored in the environment variable "query string".

3.3.3 Headers

Like most protocols, HTTP uses headers to communicate meta-data between entities. In particular, these datum govern requests and responses to and from clients and servers. The format of these headers were borrowed from MIME. And although HTTP is not a MIME compliant protocol, it uses this derived approach for communicating valuable information regarding its operation. In particular, the server must provide a MIME type for the data it provides in a response [5].

MIME Data in MIME format takes on the following form:

```
keyword: data
```

where keyword is a standard field and data is a value or setting. Some possibilities are:

```
Content-type:     text/html
Content-length:   100
Content-Language: en
Content-Encoding: ascii
```

the MIME type in the above example is "text/html". There are many MIME types available and are standardized. These types can be associated with a

filename extension[6]

When responding to a client request, a server provides a header as a series of entries of "keyword: value", with each "keyword: value" being placed on a single line, followed by a single blank line, then followed by the message itself[5].

Negotiating Compatibility HTTP headers are used by client and server in version 1.1 of the protocol to negotiate compatibility with respect to the exchange of information. There are two types of negotiations that may occur: server driven and agent driven.

Server driven negotiation is in response to a request made by a client. The client provides a set of acceptable settings and the server determines which settings are acceptable.

In agent driven negotiation, the client requests for a list of settings, and then selects the acceptable ones for the given connection.

An advantage of server driven negotiation is that it only requires a single request. An advantage of agent driven negotiation is that the client has control over the negotiation; however, this comes at a disadvantage of requiring an additional step.

HTTP deals with negotiation using what is known as "accept-headers". These headers are described in detail in [10]. Accept headers come in two flavors, the general form which deals with media types:

```
Accept=  Accept: media-range [ accept-params ]
```

and a specific form, which deals with character sets, encodings, languages, and ranges:

```
Accept-Charset:  value
Accept-Encoding: value
Accept-Language: value
Accept-Ranges:   value
```

[6] A filename is specified as a string of characters delimited by a dot, the filename extension is the exclusive string of characters to the right of the dot. This is also referred to as a filename suffix.

For example, we may have:

```
Accept: text/*;q=0.3, text/html;q=0.7, text/html;level=1,
        text/html;level=2;q=0.4, */*;q=0.5
```

or

```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

Parameters may be provided to indicate precedence to the acceptable media/values.

Conditional Requests HTTP version 1.1 provides for caching and therefore supports a mechanism whereby a client can validate its cache contents. This mechanism is called a "conditional request". There are many different types of such requests, which are issued as part of the HTTP "Request-Header" when a client communicates with a server. "Conditional requests" are concerned with "cache validators" which refer to the criteria by which a cached resource is maintained within a cache[10, 7].

For example, a client may use the following conditional request:

```
GET /index.html
If-Modified-Since: Sun 13 Nov 2000 07:00:00 GMT
```

In which case the server will respond with the file contents only if the cache validator (in this case the date and time) has been expired.

3.3.4 HTML

HTML is the primary format for data exchanged via HTTP clients and servers. Further, a web server responds to a web client using HTML when it encounters a protocol error. An HTML page is a file whose data is in HTML format, and either ends with a .htm or .html suffix. It has the following basic structure:

```
<HTML>
          BODY
</HTML>
```

Where BODY can be a set of “tags” and HTML statements as defined in the HTML language. For more information consult [1, 15, 17].

Most HTML files and there associated data files, such as graphic images are read from a static file based data store. It is possible that such pages are generated dynamically by a separate process or thread running on the server. One method by which this can be achieved is the Common Gateway Interface (CGI), which allows child processes to be created by the server in response to a web page request. In this case, the separate process and the server collaborate to provide the client with an appropriate response.

3.3.5 Errors

Errors in requests made to an HTTP server version 1.0 and 1.1, are reported to the client via a “Status-Line” and an “Entity-Body”. The “Status-Line” provides the protocol version, a status code and a descriptive message. The “Entity-Body” may contain additional information and is a valid HTML file. The file may contain information such as the protocol error code, the error name, and a brief description, the version of the server, and so on. Typically, this file is displayed by the client to the user. For example, a client requesting a resource that does not exist will get a web page similar to the following:

```
<html>
    <head>
           404 FILE NOT FOUND
    </head>
  <body>
     <p>
           404 FILE NOT FOUND
     </p>
  </body>
</html>
```

Resources belong to various classes and will be described in detail in the User Outputs section of this document. For a complete discussion, one may use [10, 7].

3.3.6 Basic TCP server

Most servers using TCP take on a similar form which can be adapted towards the implementation of different protocols. This form is similar to the following (expressed in pseudo-code):

```
getservbyname   to obtain standard port for service
gethostname     to obtain internet address for host
configure       to set parameters for socket in INET domain
create a socket
bind to the specified port
listen for client connections
enter main loop
accept a client connection
       if accept
             spawn child process or thread to handle request
```

3.3.7 Parser Main Loop

Once a connection is made a subprocess or thread is created by the server. This subprocess or thread is responsible for lexically analyzing and parsing the data being received. The tasks maybe summarized as follows (expressed in pseudo-code):

```
read from the socket
determine request
validate request
parse headers
parse request
build response or build an error message
write to the socket sending a response back to client
```

If version 1.0 of the protocol is being used, then the client-server connection is closed. For version 1.1, the connection is kept persistent until explicitly closed by either the client or the server.

The RFC governs what requests a given server will understand as well as how to respond. Much of the work required in implementing HTTP deals with being able to lexically analyze and parse the input stream in order to determine what action needs to take place.

3.3.8 MIME

For representing MIME types that a given server is internally aware of, a table will be constructed that maps file extensions to their MIME descriptions. This is needed because in order for a server to respond to the client, it must specify the content-type of the data it is providing[7] If the content-type header is not

[7]Older versions of the protocol do not support the feature.

specified, the response is in error. Further, for types that are not recognized, the content-type defaults to:

`application/octet-stream`

Most browsers will prompt the user in order to take further action.

An example of table for MIME types is as follows (expressed in C):

```
struct mime_struct {

        char *extension;
        char *mime;

}mime_tab [] =
     { "htm", "text/html" },
     { "html", "text/html" },
     { "txt", "text/plain" },
     { "gif", "image/gif" },
     { "tar", "application/x-tar" },
     /* ... */
};
```

3.3.9 Error Codes

For representing error codes, the following C structure is proposed:

```
struct error_code_struct {
              int  code;
              char *desrc;
 }error_code_tab[] =
        {200,"OK"},
        {201,"Created"},
        {202,"Accepted"},
        /* ... */
 };
```

3.3.10 Applicaiton of the Server

The HTTP server that has been developed is applicable for use in a minimal site wishing to provide basic HTTP functionality. Moreover, it is possible to extend the server and use it in custom applications that would like to utilize HTTP as an embedded communication medium for transporting data. Such an application could be a client-server application delivering information about stocks as retrieved from a database server to a HTTP enabled browser, for example. The project is not limited to a particular usuage, and can be utilized in embedded systems or server operating systems.

3.3.11 Development Environment

The software was developed freely under the Linux operating system; a free Unix-like system, developed by a loosely knit group of hackers over the Internet. The environment features all of the tools common to traditional Unix, including the GNU Compiler Collection (GCC) tools, system shell, and make. The software is implemented in ANSI conformant C++.

4 System Functional Specifications

4.1 Functional Specification

4.1.1 Functions Performed (itemized and described)

1. Initialization

   - Handle a specific set of command line options. These may include:
     - "-d" no daemon option, restricts server process to not disassociate-associate from the controlling terminal.
     - "-v" returns versioning string for the source code and the version of the protocol implemented.
   - Handle all aspects required by TCP/IP in order to properly function as server in the Internet domain. This includes obtaining the standard HTTP port via "/etc/services", and binding to the server process on that port. The TCP socket will be waiting to accept client connections after it has been initialized. Once the server binds to a port, no other process may do so. If time permits, support for the "inetd" Internet superserver maybe added, but this does not affect the operation of the server as a whole.

2. Configuration

   - Provide a mechanism to read administrator defined configuration. This will deal with parsing an input file called "/usr/local/mhttpd/mhttpd.conf" and obtaining a set of name=value pairs which are then used as parameters to the server's operation. This will allow an administrator to control the behavior of the server without explicit recompilation of the source code.

3. Error Handling

   - Provide a mechanism to handle asynchronous internal errors and log them on the host system where the server is running. From this, an administrator or user will be able to read a log file called "/usr/local/mhttpd/mhttpd.log" to view what errors happened and when they occurred. When possible, the file name, source code line and errno value will also be recorded.
   - Provide a mechanism to handle signals and server shutdown. This will work with the standard UNIX command line environment. When the server receives a signal, the default behavior will be to

shutdown the process, this will be the same for this implementation except that the server will need to do additional clean up regarding system resources that are being used. This will include shutting down TCP sockets that are being used. An administrator should be able to use the "netstat" command to verify that used sockets are left in an appropriate state after the server terminates.

- Provide a mechanism to handle synchronous errors due to the protocol. According to the RFCs, a given server must respond to certain errors with an error code and a descriptive message. This is defined as part of the protocol and will involve a response to the client. A user will be able to see the protocol response to errors through their web browser as these errors are reported via HTML. For reading the output of the server's response directly, one may telnet to the HTTP servers port and manually issue HTTP commands to see the protocol in action.

4. Protocol

   - The protocol will be restricted to version 1.0 and, will implement only GET and HEAD requests as well as CGI for GET requests as described in RFC 1945.

4.1.2 User Inputs

Due to the nature of communication protocols, it is best to express inputs and outputs to the system that are part of the protocol using Backus Naur Form[8]. This description will show the inputs as they are described in the standard[7] and as they apply to this specific implementation.

To start, the description will include the basic components, given as rules and will progress to more complicated syntactical constructs. Basic components can be expressed as lexical elements and may therefore be recognized via a finite state automata, while more complex structures are context-free grammars, and can be recognized via a push-down automata, recursive decent parser or a shift-reduce parser. [9]

[8]The description of the inputs will be done formally using a slightly modified version of BNF, as is done in RFC 1945

[9]Using tools like lex and yacc to generate finite-state automaton and shift-reduce parsers might add an investigative element to the project; although these systems may be considered "overkill", as they are typically used to implement compilers.

Basic Rules

```
OCTET   = < any 8 bit data  >
CHAR    = < any US ASCII (0-127)>
UPALPHA = < any US-ASCII upper case (A-Z)>
LOALPHA = < any US-ASCII lower case (a-z)>
ALPHA   = UPALPHA | LOALPHA
DIGIT   = <any US-ASCII digit >
CTL     = <any US-ASCII control character
                (0 - 31) and DEL (127)>
CR      = <US-ASCII CR, carriage return (13)>
LF      = <US-ASCII LF, linefeed (10)>
SP      = <US-ASCII SP, space (32)>
HT      = <US-ASCII HT, horizontal-tab (9)>
<">     = <US-ASCII double-quote mark (34)>
CRLF    = CR LF
LWS     = [CRLF] 1*( SP | HT )
TEXT    = <any OCTET except CTLs,
          but including LWS>
HEX     = "A" | "B" | "C" | "D" | "E" | "F"
        | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
word    = token | quoted-string
token   = 1*<any CHAR except CTLs or tspecials>

tspecials = "(" | ")" | "<" | ">" | "@"
          | "," | ";" | ":" | "\" | <">
          | "/" | "[" | "]" | "?" | "="
          | "{" | "}" | SP | HT

comment   = "(" *( ctext | comment ) ")"
ctext     = <any TEXT excluding "(" and ")">
quoted-string  = ( <"> *(qdtext) <"> )
qdtext         = <any CHAR except <"> and CTLs,
                  but including LWS>
```

Complex structures

1. *HTTP Version:*

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

The protocol version is defined to be a major number represented by a digit, followed by a dot, then a minor number represented by a digit. The major number has precedence over the minor number. A number such as 1.0 is less than 1.1, and a number such as 0.9 is less than 1.0.

For example,

```
HTTP/1.0
```

specifies version 1.0 of the HTTP protocol.

2. *URI:*

   This is a formal description of a URI. A URI is the generic form of a URL. In a given message exchanged by client and server, resources in HTTP are referenced by a URI as decomposed from the URL provided by the client. A server may ONLY receive a URL if it is a "proxy-server".

   A user provides the client with a URL.

```
URI            = ( absoluteURI | relativeURI ) [ "#" fragment ]

absoluteURI    = scheme ":" *( uchar | reserved )
relativeURI    = net_path | abs_path | rel_path

net_path       = "//" net_loc [ abs_path ]
abs_path       = "/" rel_path
rel_path       = [ path ] [ ";" params ] [ "?" query ]
path           = fsegment *( "/" segment )
fsegment       = 1*pchar
segment        = *pchar

params         = param *( ";" param )
param          = *( pchar | "/" )

scheme         = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc        = *( pchar | ";" | "?" )
query          = *( uchar | reserved )
fragment       = *( uchar | reserved )

pchar          = uchar | ":" | "@" | "&" | "=" | "+"
uchar          = unreserved | escape
```

```
unreserved     = ALPHA | DIGIT | safe | extra | national

escape         = "%" HEX HEX
reserved       = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra          = "!" | "*" | "'" | "(" | ")" | ","
safe           = "$" | "-" | "_" | "."
unsafe         = CTL | SP | <"> | "#" | "%" | "<" | ">"
national       = <any OCTET excluding ALPHA, DIGIT,
```

For example, we may have:

```
/student.html
```

3. *URL:*

   The URL is a specific form of URI, and is described here formally for HTTP 1.0.

```
http_URL       = "http:" "//" host [ ":" port ] [ abs_path ]

host           = <A legal Internet host domain name
                  or IP address (in dotted-decimal form),
                  as defined by Section 2.1 of RFC 1123>
port           = *DIGIT
```

   For example, we may have:

```
http://www.njit.edu/index.html
```

   Where http is the protocol, host is www.njit.edu and absolute path is /index.html. The protocol specifies that if an explicit port is not given, then port 80 is to be assummed.

4. *DATE:*

   The following describes three possible forms of date that HTTP servers recognize.

```
HTTP-date      = rfc1123-date | rfc850-date | asctime-date

rfc1123-date   = wkday "," SP date1 SP time SP "GMT"
```

```
rfc850-date    = weekday "," SP date2 SP time SP "GMT"
asctime-date   = wkday SP date3 SP time SP 4DIGIT

date1          = 2DIGIT SP month SP 4DIGIT
                 ; day-month-year (e.g., 02-Jun-82)
date3          = month SP ( 2DIGIT | ( SP 1DIGIT ))
                 ; month day (e.g., Jun  2)

time           = 2DIGIT ":" 2DIGIT ":" 2DIGIT
                ; 00:00:00 - 23:59:59

wkday          = "Mon" | "Tue" | "Wed"
               | "Thu" | "Fri" | "Sat" | "Sun"

weekday        = "Monday" | "Tuesday" | "Wednesday"
               | "Thursday" | "Friday" | "Saturday" | "Sunday"

month          = "Jan" | "Feb" | "Mar" | "Apr"
               | "May" | "Jun" | "Jul" | "Aug"
               | "Sep" | "Oct" | "Nov" | "Dec"
```

An example of these formats maybe:

```
Mon, 13, 2000 09:30:29 GMT          ; rfc1123 representation
Wednesday, 15-Nov-00 02:00:00 GMT   ; rfc1036 representation
Wed Nov 15 02:00:00 2000            ; standard C asctime()
```

5. *Character Set:* A character set in HTTP terms, refers to a specific octet to character conversion based on some mapping or table. Character sets are standardized by IANA, and the tokens here represent valid character sets that HTTP maybe aware of. The tokens uniquely identify the character set. The one important to this implementation will be US-ASCII. The others are provided here for completeness.

```
charset = "US-ASCII"
          | "ISO-8859-1" | "ISO-8859-2" | "ISO-8859-3"
          | "ISO-8859-4" | "ISO-8859-5" | "ISO-8859-6"
          | "ISO-8859-7" | "ISO-8859-8" | "ISO-8859-9"
          | "ISO-2022-JP" | "ISO-2022-JP-2" | "ISO-2022-KR"
          | "UNICODE-1-1" | "UNICODE-1-1-UTF-7" | "UNICODE-1-1-UTF-8"
          | token
```

6. *Content Encoding:* This allows a resource to be stored in an encoded, ie, compressed format. The ones defined for HTTP 1.0 are GNU zip, which is a Limpel-Ziv variation with a 32 bit Cyclic Redundancy Check (CRC) developed by Jean-loup Galley. Compress refers to Limpel-Ziv-Welch encoding.

```
content-coding = "x-gzip" | "x-compress" | token
```

7. *Media Type:*

   (a) Media types are registered with IANA, and are represented in this form, using a type followed by subtype.

   ```
   media-type     = type "/" subtype *( ";" parameter )
   type           = token
   subtype        = token
   ```

   For example, one may have a media type of:

   ```
   text/html   ; text and html subtype
   text/*       ; all text
   ```

   These where derived from the MIME standard, and are sometimes called MIME types. These types are provided on the "Content-Type" header. A HTTP 1.0 compliant server should respond with valid types for "Content-Type".

   (b) Parameters can follow the media type in the form of attribute=value pairs.

   ```
   parameter      = attribute "=" value
   attribute      = token
   value          = token | quoted-string
   ```

8. *Product Token:* These are used for identification between clients and servers.

```
product         = token ["/" product-version]
product-version = token
```

For example:

```
User-Agent: Mozilla/1.0
```

describes Netscape version 1.0.

9. *HTTP Message:* This describes the messages exchanged by the protocol. Notice this grammer is context-free. The simple request/response structures are based on version 0.9 of the protocol and DO NOT include headers. While responses are server output, they are mentioned here in the description of an HTTP-message as they are logically part of this structure. In this section simple and full requests will be described, while responses will be deferred to the output section. One may notice the orthagonal design of the protocol, where for each request component, there is a corresponding response component.

```
HTTP-message   = Simple-Request
                | Simple-Response
                | Full-Request
                | Full-Response

Full-Request   = Request-Line
                *( General-Header
                | Request-Header
                | Entity-Header )
                CRLF
                [ Entity-Body ]

Full-Response  = Status-Line
                 *( General-Header )
                 | Response-Header
                 | Entity-Header )
                 CRLF
                 [ Entity-Body ]

Simple-Request  = "GET" SP Request-URI CRLF
Simple-Response = [ Entity-Body ]
```

An example Simple-Request acceptable to the above grammer maybe:

```
GET /index.html \r\n
```

which is a client requesting the file index.html.

10. *Message Headers:* These are the headers required by the protocol in order to generate a full response or to accept a full request. Full requests/responses are part of HTTP version 1.0.

```
HTTP-header    = field-name ":" [ field-value ] CRLF

field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, tspecials, and quoted-string>
```

    Headers can be received in any order; however, it is a good practice to send as specified with General-Headers followed by response-header then entity-header.

11. *General Header:* This is a header used to describe the message being exchanged and have relevance to the request and response. Pragmas, which are implementation defined, will not be supported.

```
General-Header = Date
                 | Pragma
```

12. *Request-Line:* This particular structure refers specifically to requests being made by a client.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

13. *Method:*

```
Method   = "GET"
           | "HEAD"
           | "POST"
           | extension-method
extension-method = token
```

    As mentioned in restrictions, the methods that the implementation will focus on are: GET and HEAD methods. A client attempting to make an unsupported request to the server will receive a 501 error of "Not Implemented".

14. *Request URI:*

    ```
    Request-URI    = absoluteURI | abs_path
    ```

    This is the URI of resources being requested from a server. It is used when the resource being requested originates from the server the client has connected to. For example, if one connects to NJIT, and wants to access a local file called "students.html", the Request-URI would look like:

    ```
    /students.html
    ```

    And as part of a Request-Line:

    ```
    GET /students.html HTTP/1.0
    ```

    The absolute path component refers to a path that starts with '/'. Here '/' does not refer to the host operating system's root file system [10], but to the server's root, ie, the "document root".

    An absoluteURI is only given if the resource does not originate from the server the client is connected to, ie, in the event we are using a proxy server.

    An example absoluteURI might be

    ```
    http://www.berkely.edu/
    ```

    And as part of a Request-Line:

    ```
    GET http://www.berkely.edu HTTP/1.0
    ```

15. *Request Header Fields:* These request headers allow clients to specify additional information regarding a request.

    ```
    Request-Header = Authorization
                   | From
                   | If-Modified-Since
                   | Referer
                   | User-Agent
    ```

[10] This depends on the server's configuration, pointing the "document root" to the hosts root filesystem is a security risk

The description of these are fully defined in the RFC. Moreover, Authentication, Referer, and If-Modified-Since will not be implemented; nonetheless, they are mentioned here as they are possible inputs to the system, but will be ignored.

16. *From:* The From header specifies an Internet mail address to the server that refers to the user accessing a resource. Do to privacy issues, this header is only sent after approval by the user.

```
From          = "From" ":" mailbox
```

Where mailbox is as defined in RFC 822 [6]. For example:

```
From:  rgd1746@njit.edu
```

17. *User-Agent:* The User-Agent header specifies what client is connecting to the server.

```
User-Agent    = "User-Agent" ":" 1*( product | comment )
```

For example,

```
User-Agent: MyWebClient/1.0 libWeb/1.0
```

As an aside, when Microsoft decided to compete with Netscape using its Internet Explorer web client, the client was designed to identify itself using the same User-Agent header as Netscape, ie, "Mozilla". This was done so web servers could not distinguish between the two browsers and deny Microsoft's web browser access.

18. *Entity-Header:* These headers describe the Entity-Body, which is the resource being requested by a client or provided by a server.

```
Entity-Header  = Allow
               | Content-Encoding
               | Content-Length
               | Content-Type
               | Expires
               | Last-Modified
               | extension-header

extension-header = HTTP-header
```

For this implementation, the Allow header, used to inform of what methods apply to a resource, will not be used. Expires, Last-Modified and extension-header will also not be used. Content-Encoding, Content-Type, and Content-Length will be supported. [11]

19. *Content-Encoding:* The Content-Encoding refers to encoding applied to the Entity-Body, it does not replace the Content-Type header which actually determines the type of the data.

    ```
    Content-Encoding = "Content-Encoding" ":" content-coding
    ```

    For example,

    ```
    Content-Encoding: x-compress
    ```

    specifies that the content is in Limpel-Ziv-Welch format. For this implementation, the server will not decode such data, or encode the data, it will be expected that the data is already in this format.

20. *Content-Type:* This header specifies the media type of the Entity-Body. If this is not specified, the receiver may inspect the data attempting to determine its format, or determine its type using its file name extension. If the data is not recognized, the type is assumed to be "application/octet-stream".

    ```
    Content-Type   = "Content-Type" ":" media-type
    ```

    For example,

    ```
    Content-Type: text/html
    ```

    specifies that the Entity-Body is an html file.

21. *Content-Length:* The Content-Length is needed to specify the length of the Entity-Body.

    ```
    Content-Length = "Content-Length" ":" 1*DIGIT
    ```

[11] The Entity-Header and Entity-Body are applicable to both request and response and are therefore part of the system's inputs and outputs. They will be discussed in the input section, and will be referred to in the output section.

For example,

```
Content-Length: 200
```

specifies the length of the Entity-Body to be 200 bytes. The decimal value of the Content-Length must be greator than or equal to 0. If not specified, a client may determine the length of the Entity-Body using the length of bytes read when a connection is close. A server may not use this mechanism because it requires the connection to be closed, and therefore would not allow a server to return a response to the client. For HTTP 1.0, a Content-Lenght header must be provided on response.

If a server receives a request where it cannot determine the Content-Length, ie, the header is not provided, a 400 (bad request) error is generated.

22. *Entity-Body:* An Entity-Body is a stream of octets. This is the underlying data that is being requested or transmitted by client and server, respectively. The Entity-Header: Content-Encoding, Content-Type and Content-Length specify metainformation regarding the Entity-Body. It may be part of a request, for example, as part of a POST method, but most often is part of response.

```
Entity-Body       = *OCTET
```

Not all responses include an Entity-Body, in particular, the response to a HEAD request will not include a body, as well as status of 204 (no content), 304 (not modified). All other responses include an Entity-Body, otherwise they must specify a Content-Length of 0.

Input from the Operating Environment These inputs reflect configuration and initialization parameters specified to the server by an administrator or from a configuration, administrative file.

1. *Command Line Options* These will be the following:

```
% server_exec name -d # indicates to run as an ordinary process
% server_exec_name -v # report versioning information
```

2. *Configuration File Options* These will be specified in "/usr/local/mhttpd/mhttpd.conf" and will have the following syntax:

```
SERVER_PORT=DECIMAL
SERVER_ROOT=UNIX_PATH
```

Where DECIMAL is a decimal number greator than 1024, the reserved range of ports. UNIX_PATH is a valid path as defined for UNIX operating systems. This must actually reflect the location of the server as it is installed on the host system.

3. *TCP initialization parameters* The server will recieve as input from the "/etc/services" database the standard port to bind to in absence of an administrator specified port.

4. *Signals* The server is subject to receiving and handling signals. Because server's running on UNIX do so in the "background", traditional UNIX allows communication to these processes via the signal mechanism. For more detailed description of signals consult the UNIX programmer's manual and the "/usr/include/signal.h" header file.

4.1.3 User Outputs

Complex Structures

1. *Simple-Response:* This response does not provide headers, as a Simple-Request does not provide headers. This format is not ideal; however, it is provided to keep in line with version 0.9 of the protocol. Only a GET method maybe used with this type of request.

```
Simple-Response = [ Entity-Body ]
```

A server may simply respond with just the octet stream that makes up the resource requested.

2. *Full-Response:* A full response is defined as:

```
Full-Response   = Status-Line
                *( General-Header
                | Response-Header
                | Entity-Header )
                CRLF
                [ Entity-Body ]
```

3. *Status-Line:* This is the first line of a Full-Response message. It is made up of a protocol version followed by a numeric status code, followed by a descriptive test specifying the reason. The fields are separated via white space.

   ```
   Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
   ```

   For example,

   ```
   "HTTP/1.0 200"
   ```

   Indicates OK.

4. *Status Codes:* The status codes are classified as:

1xx	Informational, reserved for future use
2xx	Sucess, action understood and carried out
3xx	Redirection, further action needed
4xx	Client Error, Problem with a request
5xx	Server Error, Problem with a response

   The defined Status-Codes for version 1.0 are:

   ```
   Status-Code    = "200"
                  | "201"
                  | "202"
                  | "204"
                  | "301"
                  | "302"
                  | "304"
                  | "400"
                  | "401"
                  | "403"
                  | "404"
                  | "500"
                  | "501"
                  | "502"
                  | "503"
                  | extension-code

   extension-code = 3DIGIT
   ```

The Status-Codes of 201, 202 and 204 may not apply.

The 3xx class Status-Codes will not be implemented, as described in Limitations and Restrictions. This class of Status-Codes indicate redirection and are used to auto-redirect a client to a new or preferred location for a resource.

5. *Reason Phrase:* The Reason-Phrase describes textually the meaning of the status code.

```
Reason-Phrase  = *<TEXT, excluding CR, LF>
```

For example:

```
"OK"
"Created"
"Accepted"
"No Content"
"Moved Permanently"
"Moved Temporarily"
"Not Modified"
"Bad Request"
"Unauthorized"
"Forbidden"
"Not Found"
"Internal Server Error"
"Not Implemented"
"Bad Gateway"
"Service Unavailable"
```

6. *General-Header:* This was discussed in the User Inputs section of this document.

7. *Response-Header:*

```
Response-Header = Location
                | Server
                | WWW-Authenticate
```

WWW-Authentication will not be implemented as described in Limitations and Restrictions.

8. *Location:* This refers to the Location as specified in the Request-URI, unmodified.

   ```
   Location       = "Location" ":" absoluteURI
   ```

   For example:

   ```
   Location: http://www.njit.edu/student.html
   ```

   With respect to a 3xx Status-Code, the Location field indicates where the new location or preferred location is.

9. *Server:* This describes the server responding to the request.

   ```
   Server         = "Server" ":" 1*( product | comment )
   ```

   For example,

   ```
   Server: micro HTTP/1.0
   ```

10. *Entity-Header:* This was discussed in the User Inputs section of this document.

11. *Entity-Body:* This was discussed in the User Inputs section of this document.

Output from the Operating Environment

1. *Exception Log* Log files are used to report asyncronous errors to the administrator. These are errors that are encountered outside of normal processing, and are usually handled as exceptions. [12] The system will log such errors in "/usr/local/mhttpd/mhttpd.log" emitting them in the following output format:

   ```
   exception_name file_name line_number date
   ```

[12] Exceptions here refer to C++ style exceptions. Note, that in C, one can implement a similar mechanism using longjump() and setjump(). More accurately, an exception requires us to save the current state and change to a different state, including changing the program counter/instruction pointer to a different address in memory. The designated address is an entry point intended to handle the error. In C, and even more so, in Assembler, this mechanism is manually implemented by the programmer.

Where exception_name is a descriptive message indicating the error that occured. File_name is the name of the source file where the error occured. Line_number is the source line where the error occured. Date is the output of the asctime() C routine at the time the exception occured.

2. *Transaction Log* These will be a string message outputted in "/usr/local/mhttpd/mhttpd.log" and will have the following output format:

```
peer_IP_address Request-Line
```

Where peer_IP_address is the IP address of the client connection and Request-Line is the request made by the client as specified in this document and in RFC 1945.

3. *Log Message* The system will output additional messages as produced by components in the system. There format is that of US-ASCII string.

4.1.4 System Files

File	Abstract Description
/etc/mttpd.conf	server configuration files
/usr/local/mhttpd/mhttpd.log	server log files
/usr/local/mhttpd/htdocs	default document root for web pages
/usr/local/mhttpd/error	directory containing HTML for errors

4.1.5 External and Internal Limitations and Restrictions

Protocol An individual during the course of a semester would find it impossible to implement the entire HTTP protocol, therefore the project will be limited to a subset of the protocol whereas the resulting product will provide modest functionality and possible room for experimentation and research. The basis of this will be provided by RFC 1945[13], which describes the Hyper Text Transport Protocol versions 0.9 and 1.0.

1. This implementation will adhere to programming standards as set forth by POSIX, ANSI and ISO. The implementation should therefore be portable across UNIX like systems.

[13]The inputs and output section of this document was derived from RFC 1945, this author does not which to imply in any way that he is the author of the web protocol.

2. This implementation will NOT include POST, LINK, UNLINK, DELETE, PUT methods.

3. This implementation will NOT include full support for Request/Response headers, full support for Entity-Headers, full support for HTTP 1.1. [14]

4. This implementation will NOT include any authentication.

5. This implementation will NOT include support for redirection or Status-Codes 3xx. It may not include Status-Codes 201,202, and 204.

6. This implementation will NOT consider character sets other than US-ASCII.

7. This implementation will NOT consider encoding or decoding data, but will assume that if encoded data is to be transmitted, that it is already in the specified form, and therefore, only the appropriate Content-Encoding header needs to be specified.

8. This implementation will NOT provide pragmas.

9. This implementation will ONLY be aware of the following Media-Types:

```
"mp2", "audio/x-mpeg"
"mpa", "audio/x-mpeg"
"abs", "audio/x-mpeg"
"mpega", "audio/x-mpeg"
"mpeg", "video/mpeg"
"mpg", "video/mpeg"
"mpe", "video/mpeg"
"mpv", "video/mpeg"
"vbs", "video/mpeg"
"mpegv", "video/mpeg"
"bin", "application/octet-stream"
"com", "application/octet-stream"
"dll", "application/octet-stream"
"bmp", "image/x-MS-bmp"
"exe", "application/octet-stream"
"mid", "audio/x-midi"
```

[14] A possible exception to HTTP 1.1 support maybe persistent connections.

```
"midi", "audio/x-midi"
"htm", "text/html"
"html", "text/html"
"txt", "text/plain"
"gif", "image/gif"
"tar", "application/x-tar"
"jpg", "image/jpeg"
"jpeg", "image/jpeg"
"png", "image/png"
"ra", "audio/x-pn-realaudio"
"ram", "audio/x-pn-realaudio"
"sys", "application/octet-stream"
"wav", "audio/x-wav"
"xbm", "image/x-xbitmap"
"zip", "application/x-zip"
"gz" , "application/x-gzip"
```

A client making an unsupported request to the server will receive a 501 error of "Not Implemented".

Unrecognized headers will be ignored.

CGI CGI support will be provided for GET requests ONLY.

5 System Performance Requirement

5.1 Efficiency

The system is intended to be as small as possible in terms of its memory footprint, while still being functional. The compiler and the STL implementation play an important role in this; however, the use of inheritance, templates, virtual functions, and so forth, all carry additional overhead in terms of size and speed. For this reason, these features have been used where approperiate.

In terms of speed, the code uses STL containers for retrieval of data, such as std::map, with known worst case execution O(logN). The system also utilized threads, instead of processes, and thread pools instead of creating threads on demand, to minimize execution overhead.

5.2 Reliability

The system should not fail at any time. If an error occurs that is internal, it should be handled internally, and the user should receive a message either in the form of a log, or a HTML page stating that an internal server error has occured. The system should not crash at any time.

5.2.1 Description of Reliability Measures

Due to the strict reliability requirement, the system utilizes good software design techniques to alleviate errors. In addition, the system has been tested and the code has been reviewed sufficiently.

5.2.2 Error/Failure Detection and Recovery

The system has an ellaborate error detection mechanism utilizing a per-component class based exception handling mechanism. Each known error generates an exception, which is thrown along with diagnostic information regarding the error. Exceptions are caught at the top-level and either logged to a file, or represented as an HTML file with status code 500, internal server error, or both. The exceptions are represented in class hierarchies. Diagnostic information includes file and line number, along with a description of what went wrong. Not all information is presented to the user.

5.2.3 Allowable/Acceptable Error/Failure Rate

There is no acceptable error/failure rate. Every error that is created is logged for future enhancement. Errors that occur that do not prevent the system from functioning, are of lesser importance, and usually are not an indication of some extreme problem. In other words, the server should function as advertised with respect to the client making requests. Errors that are client based are not considered part of this system. For example, a client requesting a file that does not exist, is not an example of an acceptable error, because the server is functioning properly in responding with an error code of 404, file not found, and logging the transaction.

5.3 Security

Security is considered with respect to file access, the server's root directory, running the server process as user "nobody", asserting that static sized

buffers can not be overrun while handling a request, appropriate release of resources and the execution CGI scripts. Explicit support for the Secure Socket Layer WILL NOT be provided.

5.3.1 Hardware Security

There are several security measures that can be implemented in hardware. They will only be mentioned here briefly; however, it is outside of the scope of this project to consider hardware security measures for Internet based client-server applications. One obvious measure is to utilize a firewall. Firewalls maybe embedded into routing hardware and can be obtained commercially. There are also several embedded devices designed to monitor IP traffic to and from network hosts. The hardware itself should be secure with a limited number of administrators controlling it. Aside from this, the real focus on security directly related to the system, will be in software.

5.3.2 Software Security

The software is designed to be installed and removed only by "root". Explicit checks for this are made in the installation/un-installation scripts and by the operating system.

The files are installed with restricted ownership being granted to the user "nobody". At this time, the server does not access files from other users.

5.3.3 Data Security

It is the responsibility of the user to protect the integrity of the data being provided.

5.3.4 Execution Security

The system is implemented to be started by "root", privaledged user, and thereafter, assumes operation as user "nobody". This is an account with no login.

The server concatenates to all file requests the path to where it believes the files reside. That is, a request such as GET / HTTP/1.0, is transliterated to a request for a file /usr/local/mhttpd/htdocs/index.htm. In this way, external clients cannot access the root file system.

Requests for parent directories are strictly prohibited. Files are served only if they have approperiate read permissions. Files are executed only if the have approperiate execute permissions. Memory managed is done carefully, where static allocated buffers are monitored for length, to avoid buffer

overruns. In addition, the server restricts the number of threads that can be spawned, to prevent "denial of service" attacks.

In addition, any internal error that is encountered, as well as any security violation and all client requests are logged by default in system files. The system files have well defined formats, so that other tools and applications can utilize the data easily.

5.4 Maintainability

The software is devided into several directories, all, with the exception for the test executables, build into seperate libraries. The code is divided in to modules containing a class, or related classes.

5.5 Modifiability

The software can be modified easily. Great care has been taken to ensure that header files are documented to act as manual pages. Every class and class member function is described in detail in the source code. The system should be easily modifiable, and the code should be easily re-usable.

5.6 Portability

At this time, the system runs on Unix and Unix-like operating systems. Transitioning the code to other operating systems should be a matter of re-implementing certain low-level classes that encapsulate functionality dealing with the operating system. This includes initializing and running a server socket, creating threads, thread locking mechanisms, and so on.

6 System Design Overview

6.1 System Data Flow Diagrams

The system's flow charts, Figures 2 and 3, illustrate the flow of data elements throughout the functional units of the system. A description of the data flow labels is given in Table 2.

6.2 System Structure Charts

The system's organizational chart, Figure 1, illustrates the functional units of the system. A descripion of the numeric labels is given in Table 1.

Figure 1: System Flow Chart, Part 1

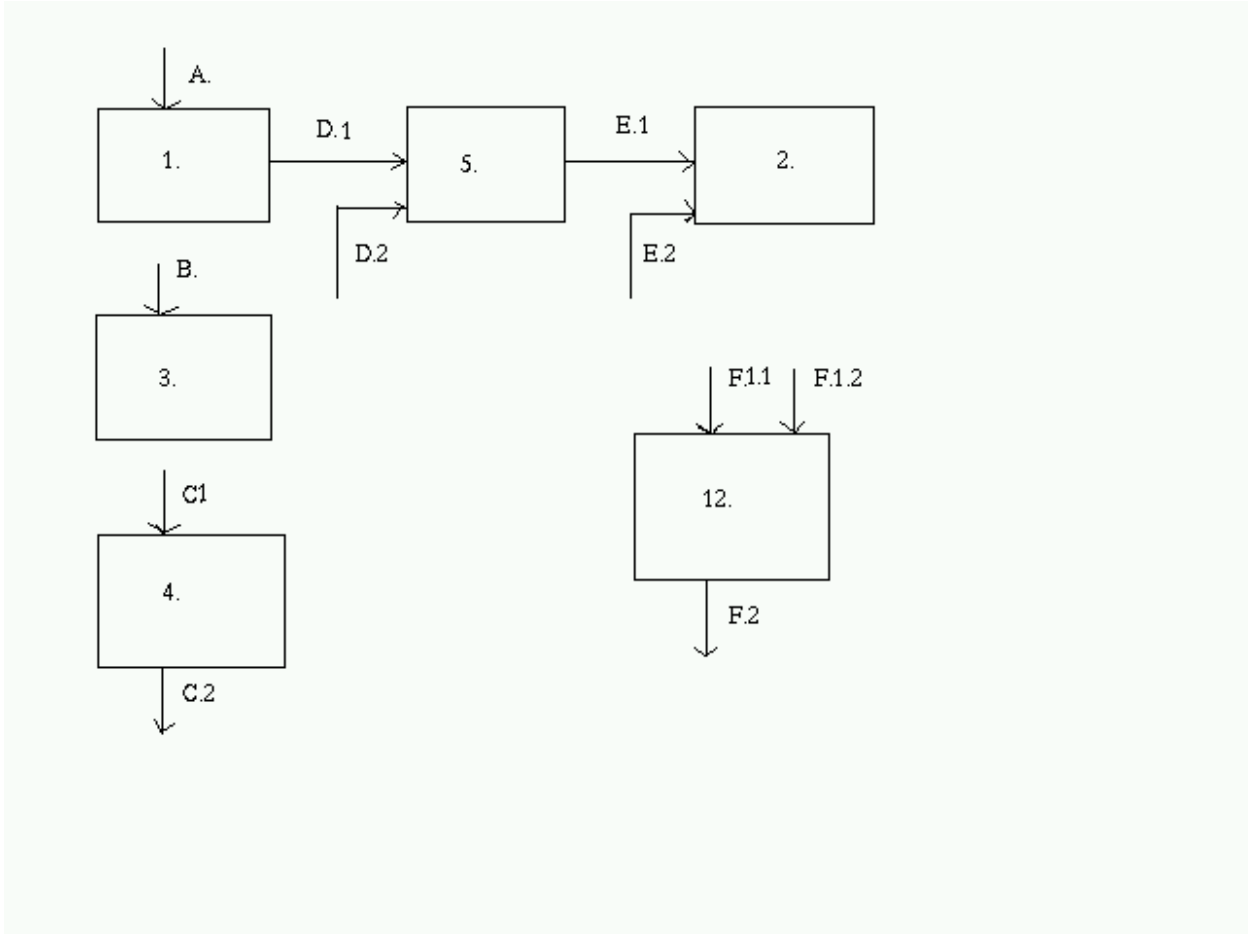

Figure 2: System Flow Chat, Part 2

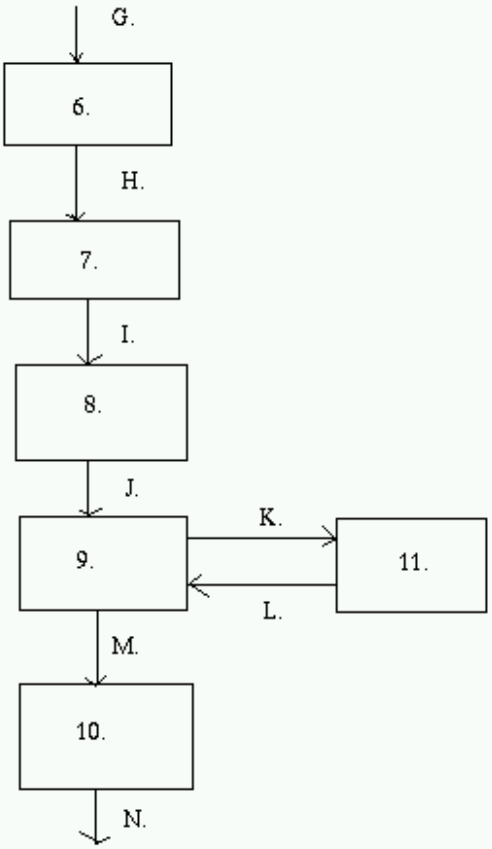

Figure 3: System Structure Chart

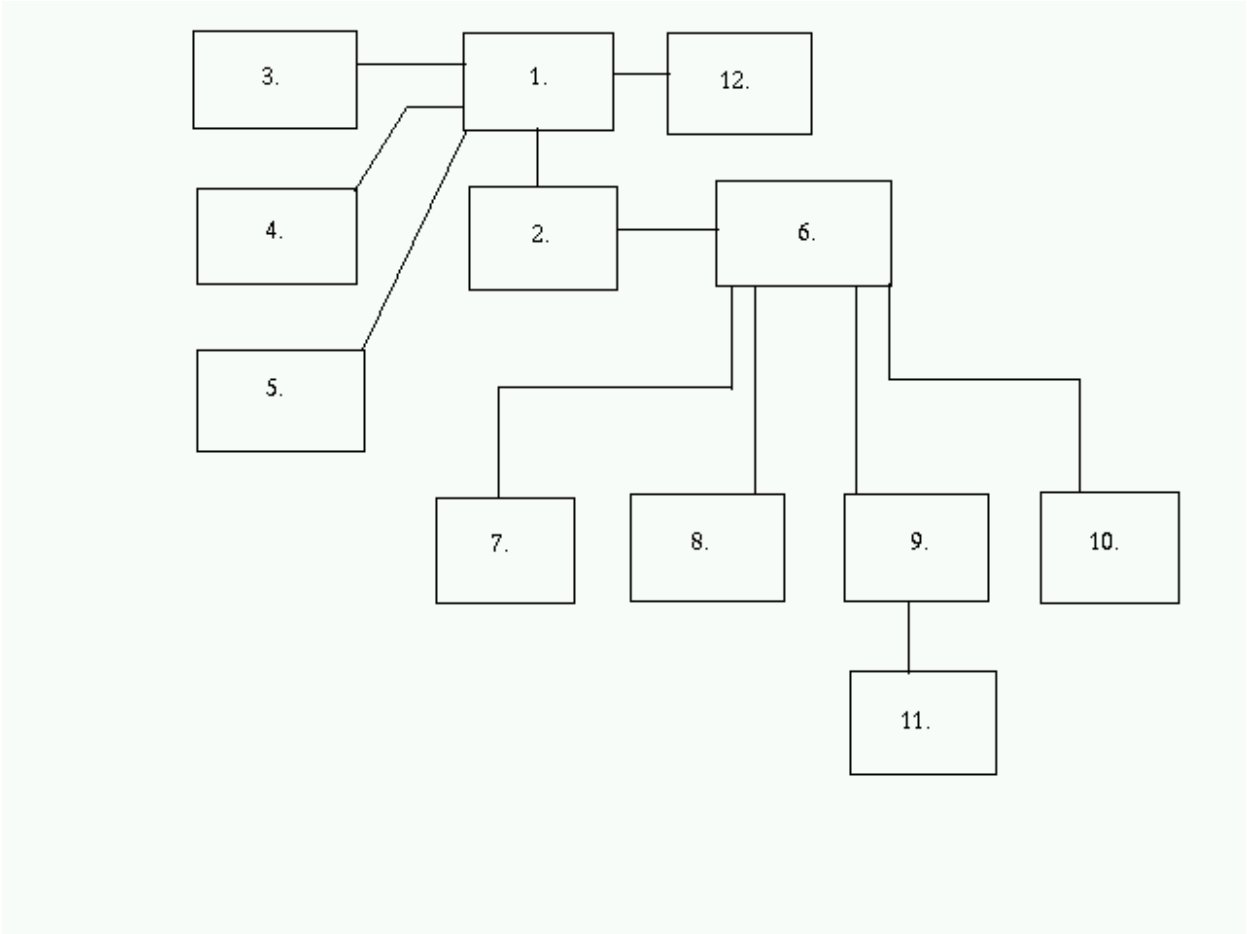

Table 1: System Organization Chart Label Definitions

1.	main
2.	initialization
3.	signal handling
4.	exception handling
5.	configuration input processing
6.	http version 1.0 support
7.	read from client
8.	parse http request
9.	produce http response
10.	write to client
11.	CGI support
12.	server log

Table 2: System Flow Chart Label Definitions

A.	command line option, unprocessed, provided via user
B.	signal, provided by operating system
C.1	exception
C.2	exception Log, output to log file
D.1	command line option, processed
D.2	configuration, provided via configuration file[15]
E.1	initialization parameters
E.2	TCP initialization parameters[16]
F.1.1	transaction message
F.1.2	log message
F.2	log message
G.	TCP connection message
H.	TCP data input, provided by client
I.	HTTP message (request), unprocessed
J.	HTTP message (request), processed
K.	CGI request
L.	CGI response, provided by external script
M.	HTTP response
N.	TCP data output

6.3 Equipment Configuration

The system will first be implemented on the Linux Operating System. [17] Once the system is tested and functioning, it will be ported onto Solaris, so that it may run on NJIT's systems.[18] In order to run, the following are prerequisite:

- The server must be started by the UNIX administrative user "root". It will then assume operation as user "nobody".
- The system on which the server is to be run must have a user "nobody" account, which is an account that has no login.
- The system on which the server is to be run must have TCP/IP configured properly.
- The system on which the server is to be run must have a valid network interface installed and configured proberly.
- The system on which the server is to be run must have a properly installed execution environment including the files discussed in "System Files". This includes the approperiate file permissions and ownership.
- A standard web brower should be available so as to be able to test the server's operation.
- A standard telnet client should be available so as to be able to issue HTTP requests manually to the server.

6.4 Implementation Languages

The language chosen was C++. The reasons for this is the following:

- C++ supports large scale software development utilizing Object Orientated facilities and Generic Programming facilities.
- C++ performance is, on the average, as good as C, which is faster than other languages that support Object Orientated facilities.
- C++ supports C and Assembler bindings. This allows for direct access to lower level calls to Standard C library routines and the Operating System.

[17] Linux is a UNIX like system written from scratch by Linux Torvalds and a loosely knit group of hackers.

[18] Porting will not be difficult due to similarities between the two systems. It will be a recompilation and test.

6.5 Required Support Software

The system has external requirements that the runtime libraries normally provided by C and C++ be available for dynamic or static linkage. Further, the underlying operating system needs to support the TCP/IP protocol suite and the Berkely Unix user-level interface to TCP/IP, commonly known as Berkely Sockets. With respect to C++, the Standard Template Library software must be available.

7 System Verification

A system with this type of functionality requires test programs to be developed to excercise key features and functionality. Further, the system should work with Web browsers, and they can be employed to test the system as well.

The test programs developed test functionality incrementally. They are listed from most trivial to most complex functionality.

7.1 Items/Functions to be Tested

The items to be tested are the following:

1. The configuration and messaging subsystem.
2. Error, exception and security log subsystem.
3. Signal handling subsystem.
4. Thread pool subsystem.
5. Basic server subsystem.
6. Basic HTTP subsystem.

7.2 Description of Test Cases

1. The configuration and messaging subsystem is relatively easy to test. There are only two logical operations that can occur: The system can either retrieve a string from a configuration file, or it can retrieve a string from internal memory. Strings can represent anything, and are defined by other components. Each string is retrieved via a component id, also known as a category id, and an integer id. The category id and

the integer id together form the primary key for retrieving the string. The test case initiliazes two types of strings, one to be read from a file, the other from in memory. For data read from files, seperate file needs to be created, containing a name=value pair[19].

2. The log subsystem is tested by opening the three log files, and writing test messages. The results maybe checked visually in the approperiate log file.

3. The signal handling subsystem extends the Unix signal handling model by providing an interface class and a Singleton class to allow signal handling to be carried out via classes. The test declares a class that implements the interface, and registers it's signal handlers. From the command line, signals maybe delivered to assert that the approperiate actions are taking place.

4. The thread pool subsystem executes tasks. The test creates two sample tasks that print the classical "hello world" message. It also includes a test for signal handling, to test signal handling along with the usuage of threads.

5. The basic server creates a server socket, and then provides a simple protocol handler that echoes back the HTTP request sent to it.

6. The HTTP subsystem test includes a basic HTTP client. The client, can provide the server with several build in requests to excercise it, or, the user, from the command line, can specify a request line and read the response from the console.

7.3 Justification of Test Cases

1. For configuration, it is important to know that system data is being retrieved correctly, and that configuration files are being parsed proberly. Without this, the server will not function proberly.

2. For Logging, it is important to know that log files are being opened successfully, and that data is being written out proberly, and with the approperiate delimeters. When the server runs as a deamon, it becomes difficult to attach to the process, even with a debugger, so it is best to assert that file logging is working so as to be used as an aid in further debugging if necessary.

[19] All configuration data is interpreted as name=value pairs.

3. For Signal handling, it is important to know that the server can register approperiately its signal handlers, as well as receive the signals. This problem is compounded with the fact that this is a multi-threaded application. Signal handling and approperiate shutdown is imperative. Testing signal handling outside of threads is only a partial test, as signals can sometimes effect normal thread operation depending on how the underlying thread system chooses to deliver the signal, and to what thread(s).

4. Threads are difficult to test by themselves. To attempt to test them in collaboration with other components is not always feasible. However, the usuage of threads is contained to running a particular task, and therefore, can easily be tested outside of the system. The only additional test included here is check how signals are handled in conjunction with threads

5. Part of the beauty of the implementation, is that it can support different protocols rather easily, with little modifications to any other code in the system. By building a simple echo server, we test the socket management code, and implicitly perform integration testing for other components.

6. HTTP client is a nice to have because it can be used to test the HTTP parser/server, as well for other projects. This is; however, a minimal socket based/command line interfaced client.

7.4 Test Run Procedures and Results

The results of all tests were successful.

7.5 Discussion of Test Results

This is an Internet server, the tests performed where for functionality. There are many other types of tests that can be run to test security, performance, and so on. These tests produce more empiracal data, and already exist. As far as the results produced with functionality testing, the server works fine.

8 Conclusions

8.1 Summary

This project yields a "Basic HTTP server", within a framework which is extensible and re-usable. Designing the system took much effort, as each component can be re-used toward other work and projects. The system itself is useful, and can be applied as-is to serving web sites. The information gained from reading and understanding the RFCs is important, as well as knowledge of the HTTP protocol.

8.2 Problems Encountered and Solved

1. Designing a system that is re-usable.
2. Understanding how TCP/IP clients/servers work.
3. Parsing HTTP/1.0 protocol.
4. Providing a successful, efficient thread model.
5. Extending Unix signals to class objects.

8.3 Suggestions for Better Approaches to the Project

Instead of implementing a server in user-space in C++, perhaps it should have been implemented in kernel-space in C.

Instead of it being a one-person project, perhaps it would have been better to have two or three students working on the project.

8.4 Suggestions for Extensions to the Project

The server is relatively small, and well documented. This allows for easy changes and customizations for specific needs. There are great deal of extensions to this project that can be conceived. Here is a partial list:

1. One could develop SQL based tools to study the data produced by the server.
2. A more complete set of HTTP requests could be parsed using LEX/YACC.
3. Additional classes could be implemented to attempt to deduce if the client is automated or human.

4. Additional HTTP functionality could be added to support additional types of request methods.

References

[1] T. Berners-Lee and D. Connolly. *Request for Comments: 1866 Hyper Text Markup Langauge*. IETF, November 1995.

[2] N. Freed N. Borenstein. *Request for Comments: 2045 Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. IETF, November 1996.

[3] N. Freed N. Borenstein. *Request for Comments: 2046 Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. IETF, November 1996.

[4] N. Freed N. Borenstein. *Request for Comments: 2049 Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*. IETF, November 1996.

[5] Douglas E. Comer. *Internetworking with TCP/IP Principles Protocols and Architectures*. Prentice Hall, forth edition, 2000.

[6] D. Crocker. *Request for Comments: 822 Standard for the Format of ARPA Internet Text Messages*. UDEL, October 1989.

[7] T. Berners-Lee R. Fielding H. Frystyk. *Request for Comments: 1945 Hypertext Transfer Protocol – HTTP/1.0*. IETF, May 1996.

[8] Alan O. Freier Philip Karlton and Paul C. Kocher. *The SSL Protocol Version 3.0 draft-freier-ssl-version3-02.txt*. Internet Engineering Task Force. (IETF), November 1996.

[9] D. Kristol and L. Montulli. *Request for Comments: 2109 HTTP State Management Mechanism*. IETF, February 1997.

[10] R. Fielding J. Gettys J. Mogul H. Frystyk L. Masinter P. Leach and T. Berners-Lee. *Request for Comments: 2616 Hypertext Transfer Protocol – HTTP/1.1*. IETF, June 1999.

[11] J. Franks P. Hallam-Baker J. Hostetler S. Lawrence P. Leach A. Luotonen and L. Stewart. *Request for Comments: 2617 HTTP Authentication: Basic and Digest Access Authentication*. IETF, June 1999.

[12] T. Bernes-Lee L. Masinter M. McCahill. *Request for Comments: 1738 Uniform Resource Locators*. IETF, December 1994.

[13] R Daniel M. Mealling. *Request for Comments: 2168 Resolution of Uniform Resource Identifiers using the Domain Name System*. IETF, June 1997.

[14] K. Moore. *Request for Comments: 2047 Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text*. IETF, November 1996.

[15] E. Nebel and L. Masinter. *Request for Comments: 1867 Form-based File Upload in HTML*. IETF, November 1995.

[16] N. Freed J. Klensin J. Postel. *Request for Comments: 2048 Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*. IETF, November 1996.

[17] D. Raggett. *Request for Comments: 1942 HTML Tables*. IETF, May 1996.

[18] Andrew S. TanenBaum. *Computer Networks*. Prentice Hall, third edition, 1996.